# Software that Learns from its Own Failures

Martin Monperrus
University of Lille & Inria
martin.monperrus@univ-lille1.fr

February 4, 2015

**Abstract**

All non-trivial software systems suffer from unanticipated production failures. However, those systems are passive with respect to failures and do not take advantage of them in order to improve their future behavior: they simply wait for them to happen and trigger hard-coded failure recovery strategies. Instead, I propose a new paradigm in which software systems learn from their own failures. By using an advanced monitoring system they have a constant awareness of their own state and health. They are designed in order to automatically explore alternative recovery strategies inferred from past successful and failed executions. Their recovery capabilities are assessed by self-injection of controlled failures; this process produces knowledge in prevision of future unanticipated failures.

## 1 Introduction

All non-trivial software systems suffer from unanticipated production failures. For instance, on March 12 2012, the Mozilla Firefox web browser has crashed 270455 times. A software failure is commonly defined as an output that does not correspond to the user's expectations [3]. In the Mozilla Firefox example, the failure is that the program closes (actual behavior) instead of rendering the requested web page (expected behavior). A failure is caused by a fault in the code or an unexpected environmental condition [3]. The classical view on software failures is to combat them with techniques for: detecting faults using static and dynamic software analysis; proving the absence of certain faults; and improving the development processes and tools to prevent the introduction of faults. Those research threads yielded fundamental advances in software engineering but failed to eradicate software failures. In this paper, I take a ground-breaking perspective on this topic: instead of being passive with respect to production failures, software systems should actively monitor, exploit and learn from them.

**The Computing World Today** Let us take the story of Pat, working on her manuscript using a text processing application. On June 26, 2014, a software

failure in her text processing document results in complete loss of the last four weeks of writing. Pat's failure is useless: the conditions in which the failure happened are unknown, the fact that Pat's failure also happened on Bob and Alice's machines is not shared, the reason for which the recovery planned by the developer failed is lost.

**The Computing World Tomorrow** All text processors of the same version would collect and share execution information. When Pat's failure occurs, detailed information is sent back to an observation server running diagnosis algorithms. The server identifies a key similarity between 17 failures. The diagnosis algorithms analyze the data and triggers fine-grain monitoring. After having collected 5 more failures and the associated detailed execution information, the server runs a learning algorithm that identifies that the code variable "plugin" is responsible for the failure. The server then synthesizes a code change as a solution. To validate the change, the system proactively injects the same failure on Alice's machine and validates that the inferred solution avoids the severe data loss. The failure of user Pat is now part of in a global adaptive and collaborative learning process, the software system (the text processor) gets better after the occurrence of the failure. Pat's failure has become useful.

**My vision is that software systems can learn from their own failures, as humans do**. By "learning", self-identifying execution and environmental patterns in which the failures occur and self-synthesizing memory and code changes so that the failures become harmless. Learning is an active process: software systems must constantly assess whether the learned healing capabilities succeed, must proactively explore alternative recovery strategies, for instance using self-injection of failures.

Recent research suggests that automatically fixing software failures is possible [11]. Weimer et al. [20], Zeller et al. [6] and others have proposed algorithms that generate valid patches to automatically repair failure scenarios. Those systems have a fundamental limitation: they work offline and not in production.

To realize my vision, I propose three research directions:

- Designing new adaptive and collaborative software monitoring systems. They will provide the input data to the learning and inference algorithms, about the software executions and failures across multiple machines.
- Inventing resourceful dynamic repair techniques that learn to self-improve. They will employ specification mining and code synthesis for achieving runtime adaptive failure recovery and exploring the recovery space.
- Characterizing proactive perturbation of software systems with injected failures. When failures become useful, one can trigger controlled failures in production for the system to learn and improve its runtime knowledge and consciousness of its environment and recovery capabilities.

## 2 State of the Art

The vision is close to the research on self-healing software [9, 8], which has also been referred to as software immune systems [16]. Now, the literature seems to prefer the term "automatic repair" (runtime repair, dynamic repair) [11].

Research on automatic software repair and self-healing systems has started in years 2000. It can be decomposed in two currents [11]. The first is called "behavioral repair" concerns repair of the system code. A famous technique is Genprog [20]. It consists of generating a patch so as to make a failing test case passing. It employs different code manipulation techniques and a kind of genetic optimization to drive the repair process. Very recent advances in this active field include DirectFix [10]. Genprog and Directfix run offline, they do not consider production failures and only work on failures for which one failing test case has been written. While the code manipulation techniques of Genprog are powerful, it misses the essential capability to be executed in production.

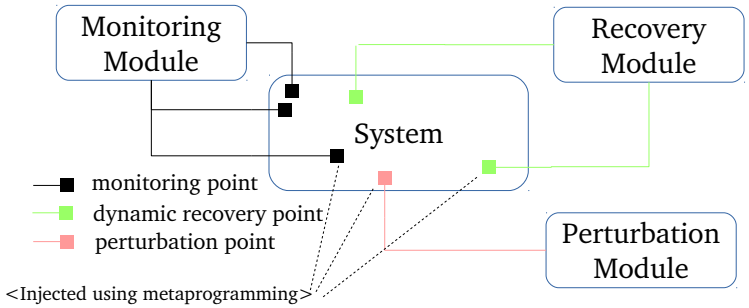
The second current of automatic software repair consists of changing the execution state at runtime. As early as 1980, Taylor and colleagues [19] introduced "robust data structures" which are able to repair their own state at runtime. More recently, Demsky and Rinard [7] proposed a similar approach for data structure repair [7]. Also much related, Perkins et al.'s ClearView [14] is a production system. Plugged into legacy x86 software, it mines runtime invariants on CPU registers and restores those invariants upon unanticipated failures. This is a strong limitation: previous experiments with reasoning on unanticipated production failures [4] has shown that real-life recovery goes far beyond invariants on CPU registers.

The closest research on this topic is by Sidoroglou et al. [15]. They have devised a system, called Assure, which is able to recover from unanticipated failures using a technique called "error virtualization". Error virtualization consists of repurposing failure handling-code for a larger class of failures. For instance, if some code has been written for handling the case where a file does not exist, it can be repurposed for handling the case where the file is not readable. Assure aims at handling unanticipated failures but it considers all failures in isolation. What is missing in Assure is to link failures together, to reason about the fact that many failures have the same cause, to understand why some recovery attempts succeed while others fail.

More fundamentally, all those systems are passive with respect to failures, they just wait for failures to happen. I propose to explore a novel direction: software systems must become active, as in "active learning". They can be active with respect to failures in a number of promising manners: by self-injecting failures to self-assess the recovery capability and by exploring alternative recovery strategies to explore the recovery space.

Fault injection can be casted in a broader concept of execution perturbation. Ammann and Knight's "data diversity" [2] consists of changing values

Figure 1: At runtime in production, the software system is augmented with a collaborative monitoring module, a resourceful recovery module and a perturbation module. All together, they form **software that learns from its own failures**.




at runtime so as to complete a computation in the presence of failures. Tang and colleagues showed that execution perturbation of floating point programs is an efficient technique to uncover failures [18]. However, nobody has ever studied automatic execution perturbation in production for constantly assessing recovery capabilities and exploring the recovery space.

# 3 Research Agenda

To construct software that learns from its own failures, there are three axes to take into account. First, software must become deeply aware of its own state and health. This can be achieved through novel automated monitoring systems. Second, software must be capable of changing and synthesizing its own recovery code at runtime. This requires completely revisiting the way recovery is handled. Third, once software has the capability to improve upon failures, it becomes ready to proactively explore its reactions to new failures, to proactively explore alternative new recovery strategies.

## 3.1 Future of Monitoring

Software that learns from its own failures needs to reason on its own internal state, health and history. To do so, it needs to embed a monitoring system. However, today's monitoring systems are not able to give an accurate view of the execution state. Let's consider the following piece of code:

```
error = login(user, password, encryptionMethod);
if (error==true) {
  log("error during login for user:"+ user);
}
```

The call to function `log` is a monitoring probe, it has been manually added by a conscientious developer to keep a trace of login errors. Now assume that

legitimate user “Bob” is not able to login due to a rare failure. The failure is due to the encryption method that is used. To be able to diagnose the failure, the system must understand that when the encryption method is “md5” it succeeds and when it is “sha1”, it fails. To do so the monitoring probe must look like the following, which has not been foreseen by the developer `log("error during login: "+context(user, encryptionMethod));` .

Upon failures, the monitoring system must collect all relevant information. One cannot only rely on manually added monitoring directives, because they are doomed to be incomplete: either calls to the monitoring system are missing, or not all variables are collected, as illustrated in the above example.

One can employ automatic injection of monitoring probes through metaprogramming, as shown in Figure 1. A monitoring system for software that learn from its failures has to meet the following requirements. First, it collects very fine grain monitoring data, up to the level of variables. This is possible because metaprogramming gives access to all elements of the code. Second, the monitoring system shall be adaptive. For instance, in nominal mode, it only collects the value of variable “user” in order to keep trace of all connections in the system. Upon the first failure, it starts to collect the other variables that are involved in the login process, so as to diagnose that the failure is due to the wrong value of “encryptionMethod”. Third, the cost of monitoring (memory, disk, bandwidth) is dynamically controlled. For instance, if disk space is limited, the monitoring system decides to only collect one out of two calls to function “login”. This is a tradeoff between keeping the core functionalities up and the speed of diagnosis. Fourth, monitoring must be collaborative. The same failure happens on many machines running the same software application. The failure information on machine $x$ can be cross-compared with the same failure happening on machine $y$. The comparison of similarities and differences drives the diagnostic.

> I envision a monitoring system that is adaptive, collaborative and resource-aware. Nobody has ever put those three capabilities together.

When monitoring becomes fine-grained and collaborative, it may become a threat to the privacy of the application’s user. Special care must be taken so as to protect the end users’ privacy. Anonymization and sampling have to be used so that no malicious attacker can exploit the system to track, spy or steal the users.

**Barriers:** The resources taken by monitoring compete with the ones needed for the code functionalities (disk, memory and bandwidth). Also, there is a major size issue: there are thousands of machines that cooperate to share monitoring data. This requires an architectural blueprint as well as core optimizations that are beyond the state of the art.

### 3.2 Acting upon Unanticipated Failures

In today's software systems, the mainstream way to communicate and handle failures is exception handling. In software design books, entire chapters are dedicated to discussing good and bad practices on exception handling [5]. For instance, the Hadoop distributed filesystem does most of its failure-handling using exceptions through 9888 catch block. Briefly, classical exception handling works as follows:

```
// failure detection
if (file==null) { // failure condition
  throw new InputException();
}

// failure handling
catch (InputException) { // recovery condition
  defaultText = "foo␣bar"; // recovery strategy
}
```

One sees the three main components of failure handling using exceptions: 1) the failure condition (here `file==null`) encodes a predicate on the system state whose value encodes the presence of a failure; 2) the recovery condition states when a failure can be handled (here when an exception of type `InputException` arrives at this location); 3) the recovery code repairs the system state it is the content of the catch block (here `defaultText = "foo bar"`). Today, all those three components are manually written and hard-coded. This is a fundamental limitation.

In today's open-ended systems, the kinds of failures and recovery are open-ended. As said above, the Eclipse development environment runs on at least 5 millions different machines. Can the developer foresee all possible failures? Can she hard-code all possible failure conditions, recovery conditions and recovery strategies? No.

I claim that that all those three components must become adaptable instead of being manually hard coded, as shown in the following example:

```
// failure detection
if (failureDetected(file, user)) {
  throw new InputException();
}

// failure handling
catch (failureRecoverable(exception, context)) {
  recover(defaultText,output);
}
```

In this simplifying example, the main changes compared to the previous one is that the failure detection condition becomes the result of the evaluation of a function, so do the recovery condition and the recovery code. By replacing

an hard-coded condition with a function evaluation, the failure-handling can be become resourceful and capable of handling unanticipated failures. In this context, the notion of "resourceful" is close to that of Abbott [1] and means two things: first, resourceful failure-handling code can be changed at runtime if previous evaluations failed to heal the system, second it can act upon an arbitrary number variables of the system, incl. those that have not been foreseen by the developers. In the example, although the manually written failure condition only involves variable "file", the correct failure condition detection may actually involve variable "user". This point also holds for the recovery condition and the recovery code since the developer cannot perfectly foresee the variables required for recovery in all cases.

This vision poses a number of challenges. First, it must be non-conflicting with the existing failure handling code (the recovery must gracefully augment the 9888 existing catch blocks in the case of Hadoop). Second, it must work in intimate collaboration with a monitoring system so as to reason on when and how a resourceful algorithm must be run to synthesize a new failure condition, recovery condition or recovery strategy. Third, it must be compatible with existing compile-time and runtime support for exceptions (not all languages support parameterizing the catch exception by function evaluation as described in our example).

One enabling solution is the introduction of runtime evaluation and first class execution objects at all stages of the failure-handling process: first failure recovery strategies can then be reused even in unanticipated cases (as opposed to only triggered in the specified cases); second failure detection recovery can be adapted: when recovery fails, alternative solutions in a recovery space are explored, for instance by taking into account new portions of the system state (new variables). Last but not least, the recovery itself can be seen declaratively: the call to `recover(defaultText,output)` searches for a solution to a recovery problem, with a declarative specification of the recovery problem (given a failure detection condition, a recovery condition and a necessary recovery post-condition).

> I propose to consider recovery under three novel angles: recovery should be the result of a synthesis problem; recovery should be a first-class execution runtime object with full intercession capabilities; recovery should become a search problem with alternative and competing solutions.

**Barriers:** The main barrier for achieving resourceful recovry is semantics. As shown in our empirical study [13], recovery uses the full range of programming language constructs with rich and complex semantics. Also, the diversity of failure conditions and recovery strategies may be too large for keeping a complete record of all encountered failures and their solutions, so as to ensure the automatic exploration of new portions of the search space.

### 3.3 Constant Assessment of Recovery Capabilities with Injected Perturbations

Once one has changed the perspective on software failures, once one considers that software failures are an opportunity for the system to self-improve, this opens radically new research directions. Today's research devises systems that are passive with respect to failures in production: they only handle failures that "naturally" happen, where "naturally" means being triggered by an external and uncontrolled cause. I envision systems that proactively perturb the execution with injected failures. The injected failures are carefully qualified with respect to the expected recovery contracts. Let us consider a concrete example.

According to our statistics on the Internet, the most common failures in Java software are null dereferences ("null pointer exceptions"). Null dereferences cause desktop, server and mobile applications to crash on a daily basis. Now, let's consider a software system built using perturbationapplied to null dereferences. The system embeds an advanced monitoring system as described above, as well as resourceful recovery for null dereferences. As such, the system has already overcome 15 unhandled null dereferences. Now, the system is augmented with a module that selectively injects null values in memory. This system would inject $x$ (say 3) null values per day so as to 1) assess that the system does not crash upon null dereferences, and 2) validate synthesized recovery pro-actively.

On the hardware side of production systems, this idea is already applied. For instance, datacenters are regularly subject to power cut so as to assess whether the alternative sources of power are up and running. I propose to explore this disruptive idea on the software side, to constantly assess whether the embedded software recovery code well handles software failures.

This can be seen as an application of the scientific method to failure recovery. The scientific method states that all hypotheses must be experimentally validated using falsification experiments. A recovery capability is also an hypothesis: if an event of type $x$ happens, the system is able to survive. By injecting an event $x$ and assessing successful failure-recovery, one ensures the truthfulness of the recovery hypothesis in production. In biology, "hormesis" refers to the positive response of biological systems (e.g. a cell) in response to a stressor. This can be seen as the exploration of the notion of hormesis in the domain of software systems. Hormesis is close to antifragility [17] and to this extent, this paper further explores the engineering of antifragile software [12].

This is different and complementary from performing failure analysis in testing phase. Failure analysis during testing enables to validate specific, well formed, small failures. However, production systems are too big and too interconnected to be reproduced in a controlled testing environment. This results in unanticipatable situations and behaviors. Failure injection in production aims at tackling those unanticipatable, untestable failures that necessarily happen in production.

Software systems must become active with respect to software failures: they must constantly assess their own recovery capabilities with failure injection, they must constantly explore the recovery space based on the injected failures. The injected failures are carefully controlled to both maximize the knowledge gained from each injection and to mitigate their impact and cost.

**Barriers:** This research direction has never been explored before. The idea of injecting faults in production is fundamentally disturbing. We do not know whether one can fully control the impact of injected faults, and whether the benefits obtained with failure injection (better failure recovery; improved knowledge) outperforms the losses (unfilled requests, unsatisfied users).

# 4 Conclusion

In this paper, I have sketched vision of software that learns to self-improve from its own failures. To realize the vision, I set up a research agenda in three points: devising the next-generation of collaborative and adaptive monitoring systems, inventing a new generation of recovery based on code synthesis and automated exploration of the recovery space, and characterizing failure injection in production. The last point is radically new and very promising: fault-injection in production is the only way to proactively assess the recovery capabilities and to proactively explore the recovery space.

# References


[1] Russell J Abbott. "Resourceful systems for fault tolerance, reliability, and safety". In: *ACM Computing Surveys (CSUR)* 22.1 (1990), pp. 35–68.

[2] P.E. Ammann and J.C. Knight. "Data diversity: an approach to software fault tolerance". In: *IEEE Transactions on Computers* 37.4 (1988), pp. 418 –425.

[3] Algirdas Avizienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004).

[4] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. "Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions". In: *Information and Software Technology* (2014).

[5] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reuseable .NET Libraries*. Addison-Wesley, 2008.

[6] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. “Generating Fixes from Object Behavior Anomalies”. In: *Proceedings of ASE*. 2009.

[7] B. Demsky and M. Rinard. “Automatic detection and repair of errors in data structures”. In: *ACM SIGPLAN Notices* 38.11 (2003), pp. 78–95.

[8] Angelos D Keromytis. “Characterizing self-healing software systems”. In: *Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. 2007, pp. 22–33.

[9] Philip Koopman. *Elements of the Self-Healing System Problem Space*. Tech. rep. Carnegie Mellon University, 2003.

[10] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “DirectFix: Looking for Simple Program Repairs”. In: *Proceedings of ICSE*. 2015.

[11] Martin Monperrus. “A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair”. In: *Proceedings of ICSE*. 2014.

[12] Martin Monperrus. *Principles of Antifragile Software*. Arxiv 1404.3056. 2014.

[13] Martin Monperrus et al. *Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages*. Tech. rep. Laboratoire d’Informatique Fondamentale de Lille, 2014.

[14] Jeff H. Perkins et al. “Automatically Patching Errors in Deployed Software”. In: *Proceedings of SOSP*. 2009.

[15] S. Sidiroglou et al. “Assure: Automatic Software Self-healing using Rescue Points”. In: *Proceedings of ASPLOS*. 2009.

[16] S. Sidiroglou et al. “Building a reactive immune system for software services”. In: *Proceedings of the USENIX Annual Technical Conference*. Vol. 161. 2oo5,(6). 2005.

[17] Nassim Nicholas Taled. *Antifragile*. Random House, 2012.

[18] Enyi Tang et al. “Perturbing Numerical Calculations for Statistical Analysis of Floating-point Program (in) Stability”. In: *Proceedings of ISSTA*. 2010.

[19] David James Taylor, David Ernest Morgan, and James P Black. “Redundancy in data structures: Improving software fault tolerance”. In: *IEEE Transactions on Software Engineering* 6 (1980), pp. 585–594.

[20] Westley Weimer et al. “Automatic program repair with evolutionary computation”. In: *Communications of the ACM* 53.5 (May 2010), p. 109.